Computer Science, Texas A&M University

Vortex: Extreme-Performance Memory Abstractions for Data-Intensive Applications

Carson Hanel, Arif Arman, Di Xiao, John Keech, and Dmitri Loguinov

Internet Research Lab (IRL)
Department of Computer Science and Engineering
Texas A&M University, College Station, TX, USA 77843

March 19, 2020

Agenda

- **Introduction**
- Motivation
- Producer Consumer
- Partitioning and Sorting
- Experiments

Introduction

- Streaming is a commonly employed paradigm for data-intensive computing
  - Often, traditional streaming applications and software packages are unsuited for extreme performance, or rates close to the speed of hardware
  - Moreover, data streaming continues to offer the same block-based communication model of the 1950s
  - Because of this, programmers must choose between "fast, but complex" (e.g., hand-tuned assembly), and "simple, but slow" (e.g., Apache Hadoop) solutions for large problems

Introduction

- Many applications in data analytics, information retrieval, and cluster computing process massive amounts of information
- In this paper we introduce the Vortex programming model, which has the following goals:
  – Offer a *simple abstraction* for larger-than-RAM inputs
  – Squeeze *maximum performance* out of hardware
- Usually, these are conflicting goals, but we show that this does not have to be the case
  – Vortex leverages access violations to create the illusion of an infinite buffer in user space
  – It is by far the simplest to use and fastest platform for various streaming workloads

Agenda

- Introduction
- **Motivation**
- Producer Consumer
- Partitioning and Sorting
- Experiments

Motivation: Coding Simplicity

- Consider the task of finding a user-defined string in a long (e.g., 32 TB) stream of data using *strstr()*
  – The stream could be originating from a disk, another thread, or arriving in real time from the network
- Traditional block-based solution:

```
Search (char* str, uint64 size, uint64 blockSize)
          strLen = strlen(str); buf = new char[blockSize];
          pos = 0; bufStart = 0;
          while (not end of data) do
                    size = blockSize – 1 – pos;
                    bytes = GetNextBlock(buf + pos, size);
                    buf [pos + bytes] = NULL;
                    if ((ptr = strstr(buf, str)) != NULL) then
                              return ptr – buf + bufStart;
                    bufStart += bytes + pos – (strLen – 1);
                    pos = strLen – 1;
                    memcpy(buf, buf + blockSize – 1 – pos, pos);
```

  – Error-prone pointer calculations
  – Memcpy() for data crossing block boundaries
  – Tedious coding practice, slow development

Motivation: Coding Simplicity

- Instead, we would like a much simpler memory abstraction that allows treating streams as infinite:

```
Thread Producer (char* buf, uint64 len)
          memset(buf, ‘a’, len);
          buf [len] = NULL;

Thread Consumer (char* buf)
          return strstr(buf, “Hello World!”);
```

- Ideally, this abstraction would provide:
  - Coding simplicity
  - No Memcpy() or boundaries
  - No error-prone pointer management
  - Complete transparency, including synchronization
  - Ability to make large (e.g., 32 TB) memset()/strstr() calls

Motivation: Faster Iterator Abstractions

- Consider implementing a producer-consumer pipeline between threads

```
Thread Producer (Iterator* it, uint64 len)
          for (i = 0; i < len; i++) do
                    it->Write(i);

Thread Consumer (Iterator* it, uint64 len)
          for (sum = 0, i = 0; i < len; i++) do
                    x = it->Read(); sum += x;
```

  – Commonly, this is done with an iterator abstraction
  – Iterators greatly reduce programming effort
  – Error-prone block management is abstracted away

Let’s look further into iterators!

Computer Science, Texas A&M University

Motivation: Faster Iterator Abstractions

```
Iterator::Read()
          x = bufR [posR];
          if posR == blockSize – 1 then
                    empty.push (bufR);
                    bufR = full.pop ();
                    posR = 0;
          else
                    posR++;
          return x;
```

Iterator Internals

```
Iterator::Write(int x)
          bufW [posW] = x;
          if posW == blockSize – 1 then
                    full.push (bufW);
                    bufW = empty.pop ();
                    posW = 0;
          else
                    posW++;
```

- Iterators exhibit non-trivial overhead
  - Writer: **3 loads** and **3 stores**
  - Reader: **4 loads** and **2 stores**
- An optimal solution requires **1 load** and **1 store**
  - Iterators thus unnecessarily stress the L1 cache, which can become a huge bottleneck in certain applications

Motivation: Faster Iterator Abstractions

- The desired abstraction would allow memory to be processed uninterrupted (i.e., without boundaries or explicit synchronization)

```
Thread Producer (int* buf, uint64 len)
        for (i = 0; i < len; i++) do
                buf [i] = i;

Thread Consumer (int* buf, uint64 len)
        for (sum = 0, i = 0; i < len; i++) do
                sum += buf [i];
```

May be larger than RAM (e.g., 32 TB)

- Benefits of this approach:
  - Requires **1 load** and **1 store** per item
  - Depending on CPU, may be 2-4x faster than an iterator
  - Regular pointers are abstracted as being “infinite” (i.e., not constrained by physical RAM)
  - Could help to maximize application throughput

Motivation: Non-Counting In-Place Partitioning

- Consider partitioning n keys across k arrays (e.g., during radix sort)
  - The size of each output buffer is unknown a-priori, which generally requires a counting pass to pre-allocate buffers
  - Key movement either requires 2n + O(1) memory or needs slow iterator abstractions
- It is desirable to eliminate these constraints and
  - Distribute the keys without the histogram pass
  - Operate in-place (i.e., using n + O(1) total memory)
  - Achieve close to optimal speed

Agenda

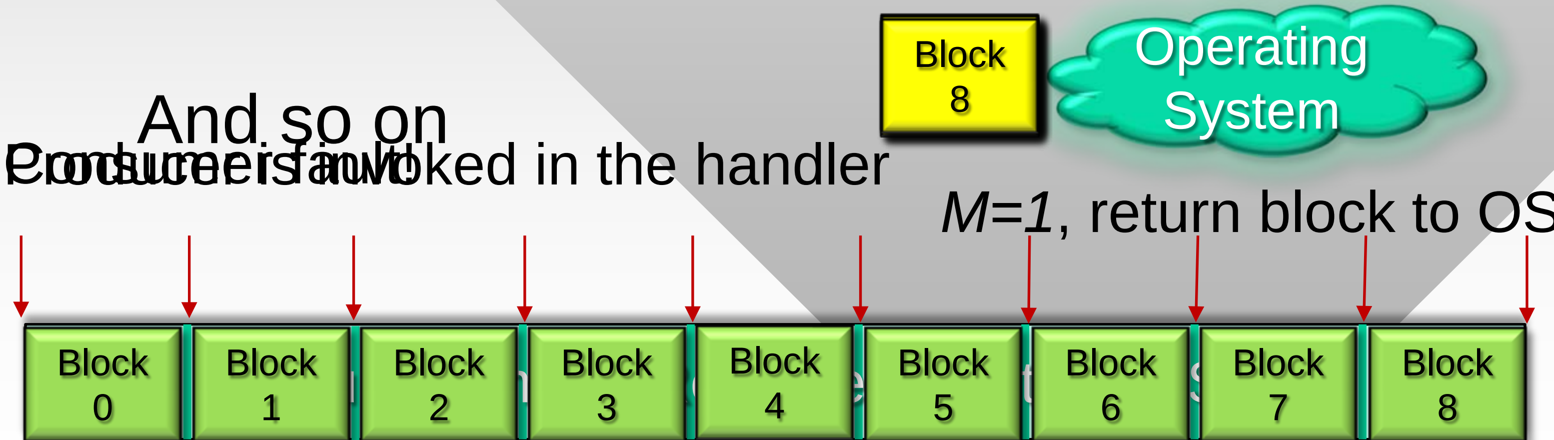
- Introduction
- Motivation
- **Producer Consumer**
- Partitioning and Sorting
- Experiments

Virtual Memory in User Space

- General idea behind Vortex
  - Access to reserved, uncommitted virtual memory generates a page fault
  - These faults result in exceptions that can be caught by a user-space handler
  - We can thus cause controlled, sequential-access violations in virtual memory
  - To fix the violation, we map physical pages to the location of the fault in the stream
  - Once the memory is available, we transparently restart the read/write instruction that caused the fault

Vortex-A

- To avoid faulting per 4-KB page, operations proceed in units (blocks) of size B (e.g., 1-2 MB)
  - To allow out-of-order reads, let M be the consumer comeback, i.e., the number of blocks by which it can return to reprocess the data
- Threads are synchronous - the producer is invoked per-block within the fault handler



Vortex-A

- Drawbacks of this model:
  - The abstraction is non-transparent to the producer thread, and thus incoming data must still be produced in block-sized increments rather than continuously
  - Threads are necessarily synchronous, as the producer is only invoked once the consumer encounters a fault
  - Consumer comeback is handled by *M*, but producer comeback has no such accommodation

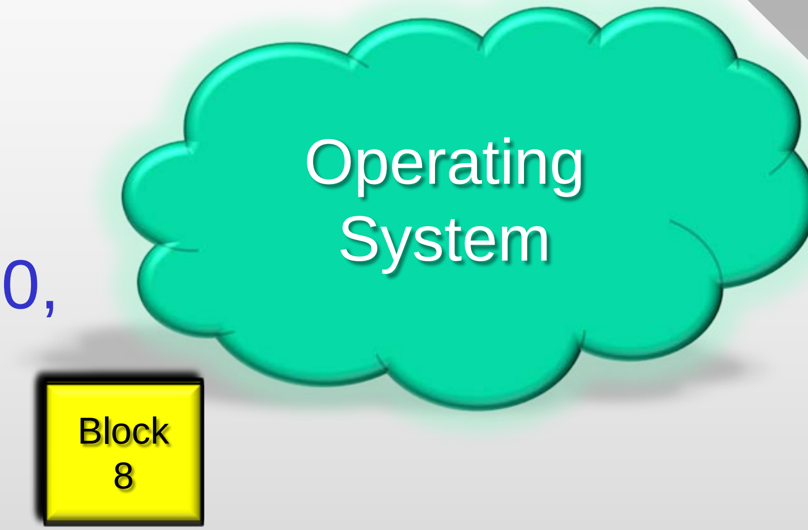
Vortex-B

- In this model, the producer is not aware of the existence of an underlying stream
  - Instead, the producer writes into an infinite buffer
  - Adds producer write-ahead N and comeback L control
- Threads are asynchronous
  - Achieved by tracking and limiting the consumer via guard pages, which cause access violations
- Employs the classical bounded producer-consumer solution to track empty and full blocks

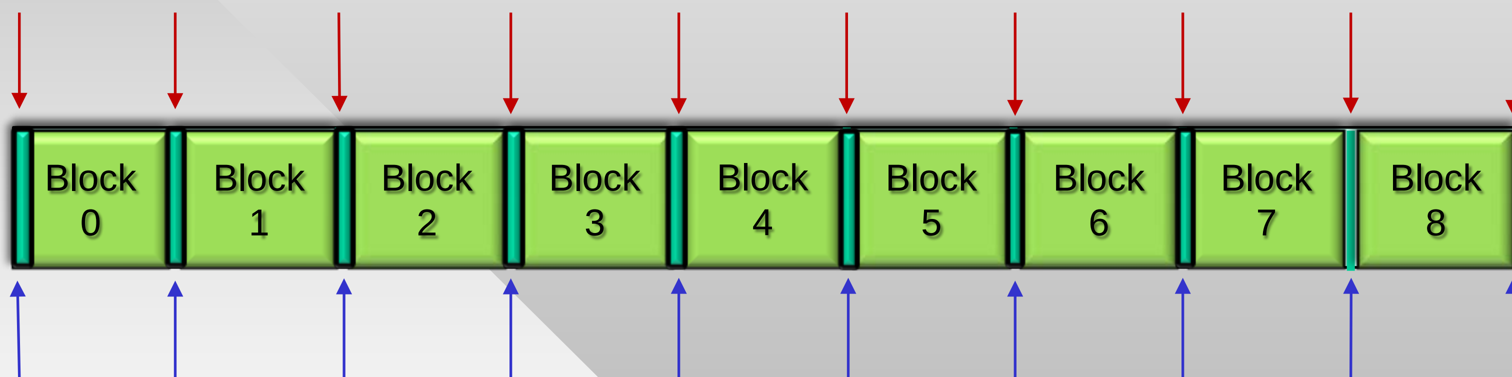
Let's see it in action!

Vortex-B

- Operation with M = 0, L = 0, N = 2



Consumer faults on a guard page



Producer fault!

M=0, return LRU block to OS

And so on

The producer installs a guard page and must wait for the consumer N=2

Vortex-B

- **Drawbacks of this model:**
  - Blocks cannot be safely consumed until they are protected by guard pages, and thus the minimum distance between threads is the full size of a block
  - Producer and consumer threads share a virtual buffer, making it more difficult to isolate them (e.g., forward consumer jumps are not supported)
  - Instead of maintaining a pre-allocated stack of blocks, memory is obtained from and released to the OS, which incurs a severe performance penalty

Vortex-C

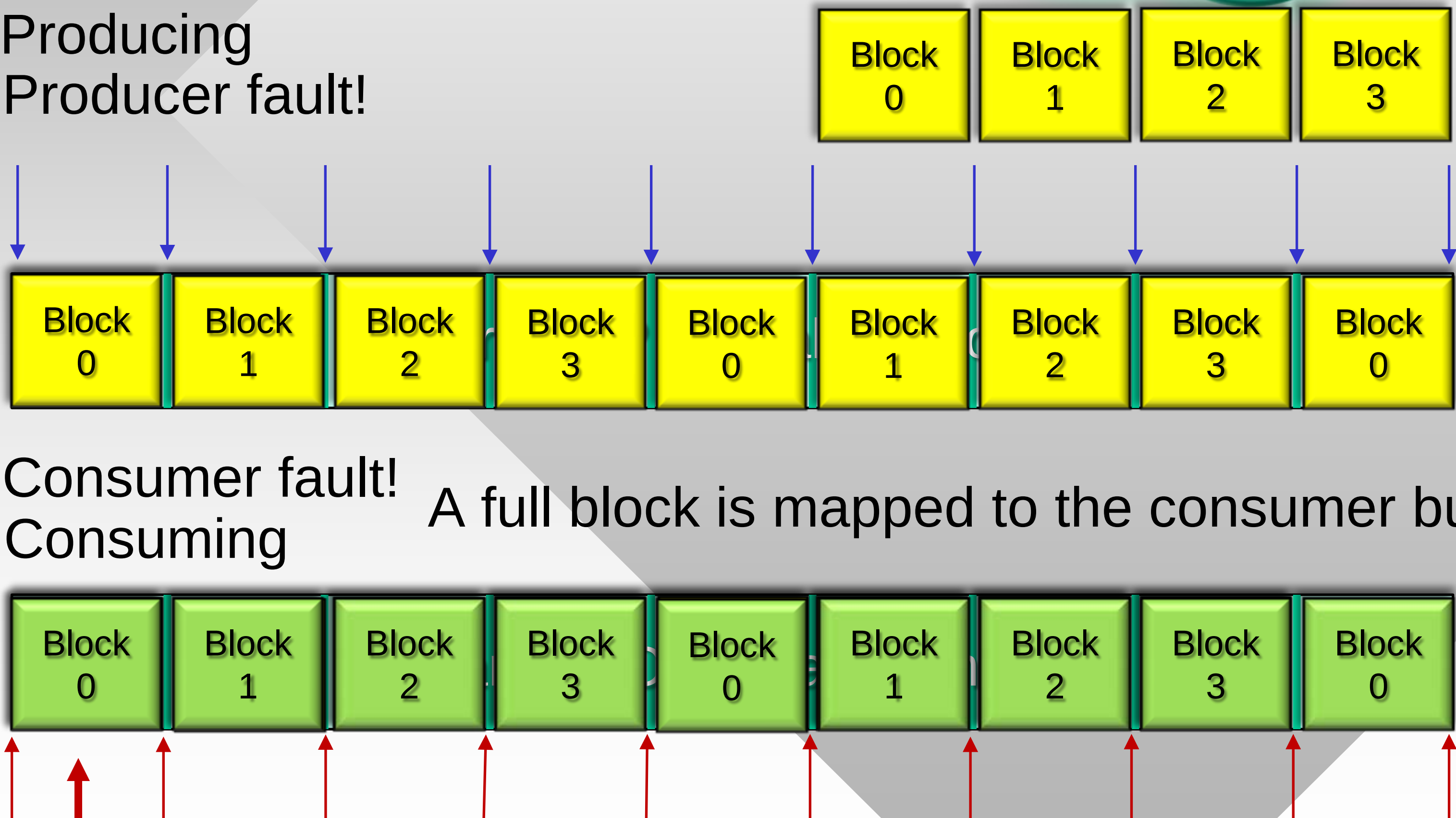
- Further improves upon **Vortex-B**
  – Pre-allocates physical memory at the start of the program instead of during runtime
  – Unlike the previous models, gains speed by retaining blocks for remapping rather than freeing to the OS
- Instead of using guard pages to track the consumer, this method uses dual-buffers
  – Threads get separate virtual-memory buffers for runtime address space isolation
  – Blocks are quickly remapped between streams

Let's see it in action!

Vortex-C

- Operation with M = 1, L = 1, N = 1

Producing
Producer fault!



Consumer fault!
Consuming

A full block is mapped to the consumer buffer

And so on with the Vortex

The consumed block is then remapped to the producer

Agenda

- Introduction
- Motivation
- Producer Consumer
- **Partitioning and Sorting**
- Experiments

Partitioning and Sorting

- We adapt Vortex to create a novel variant of **bucket sort** which utilizes:
  - Non-counting data partitioning to avoid a histogram pass
  - In-place data shuffling to stream sort with n + O(1) RAM
- To achieve non-counting data partitioning
  - Each sort bucket is reserved to the full size of input
- The result is the first in-place streaming radix sort
  - Posts a 2-4x performance improvement over prior work
  - Provides out-of-place speeds with in-place operation
- Finally, this abstraction does not require specialized code or memory management to achieve in-place sorting, being instead totally transparent

Agenda

- Introduction
- Motivation
- Producer Consumer
- Partitioning and Sorting
- **Experiments**

Experiments

Available Test Configurations			
Hardware	c_1	c_2	c_3
CPU Platform Test drive	Intel 3930K Sandy Bridge 24-disk RAID	Intel 4930K Ivy Bridge 24-disk RAID	7820X Skylake-X M.2 SSD

Experiments

File I/O Speed (MB/s)						
Framework	c_1		c_3		CPU	RAM
	Read	Write	Read	Write		
std::fstream	43	88	51	140	8%	2 MB
Win. MapViewOfFile	69	147	1,161	*	8%	32 GB
Linux mmap	1,892	1,170	1,917	641	3%	30 GB
Vortex-A	2,235	1,547	1,272	651	8%	5 MB
Vortex-B	2,231	2,394	3,211	650	8%	5 MB
Vortex-C	**2,238**	**2,399**	**3,266**	**674**	**1%**	**5 MB**

Vortex-C is 1.7x faster than mmap

Batched Producer-Consumer Rate (GB/s)							
Framework	Two core			All cores			RAM
	c_1	c_2	c_3	c_1	c_2	c_3	
Apache Storm	1.7	1.4	2.4	11.1	9.5	12.8	1.6 GB
Naiad	2.7	3.1	4.4	7.4	7.9	13.1	65 MB
Queue of Blocks	6.4	7.3	11.4	17.1	16.5	24.8	24 MB
Vortex-B	4.3	4.4	4.6	5.1	5.2	3.9	9 MB
Vortex-C	**13.5**	**16.4**	**23.3**	**38.3**	**38.4**	**65.4**	**9 MB**

Vortex-C is 5-10x faster than Storm

Computer Science, Texas A&M University

Experiments

Populating an 8 GB Vector on c_3 (GB/s)

Framework	Memory	
	Untouched	Pre-Faulted
std::vector	0.7	-
RUMA rewired vector, 4 KB pages	-	5.3
RUMA rewired vector, 2 MB pages	-	14.3
Chained Blocks	6.8	18.8
Vanishing Array (Vortex-S)	**25.1**	25.1
Static Buffer	8.0	**28.5**

Vortex-S is 3.1x faster than an untouched static buffer

Vortex-S reaches 88% of static buffer speed

Partitioning Speed of 8 GB on c_3 (M keys/s)

Framework	Write Combine	k=256	k=512
2-pass	N	339	322
Chained blocks	N	450	413
Vortex-S	N	**492**	**445**
Pre-allocated buckets	N	**509**	**464**
2-pass	Y	364	344
chained blocks	Y	461	449
Vortex-S	Y	**607**	**523**
Pre-allocated buckets	Y	**637**	**567**

Applied to partitioning, Vortex-S achieves 92-96% static buffer speed

Experiments

Fastest In-Place Radix Sorts (M keys/sec)

Sort Type	Year	8 GB of keys			24 GB of keys		
		c_1	c_2	c_3	c_1	c_2	c_3
MSB Radix	2014	19	23	26	18	21	26
MSB Radix	2017	**24**	**25**	**32**	24	26	32
MSB Radix	2019	17	19	26	**25**	**30**	**39**
Vortex Sort	**2020**	**71**	**84**	**127**	**68**	**80**	**121**

Fastest Out-Of-Place Radix Sorts (M keys/sec)

Sort Type	Year	8 GB of keys			24 GB of keys		
		c_1	c_2	c_3	c_1	c_2	c_3
LSB Radix	2011	25	25	39	Not enough RAM		
LSB Radix	2014	24	26	42			
LSB Radix	2016	19	23	34			
MSB Radix	2017	25	29	41			
MSB Radix	2017	**44**	**58**	**67**			
Vortex Sort	**2020**	**71**	**84**	**127**	**68**	**80**	**121**

Speedup Factor of Vortex-S

Compared to	8 GB			24 GB		
	c_1	c_2	c_3	c_1	c_2	c_3
Best in-place	2.9	3.3	4.0	2.7	2.7	3.1
Best out-of-place	1.6	1.4	1.9	∞		

Vortex is 2.9-4.0x faster at sorting 8 GB than the nearest in-place radix sort competitors

Vortex is 2.7-3.1x faster at sorting 24 GB than the nearest in-place radix sort competitors

Even considering out-of-place sorts, Vortex is still 1.6-1.9x faster at sorting 8 GB, and can run sort sizes twice as large

Thank you!

Any questions?

***Contact*: Carson@cse.tamu.edu**